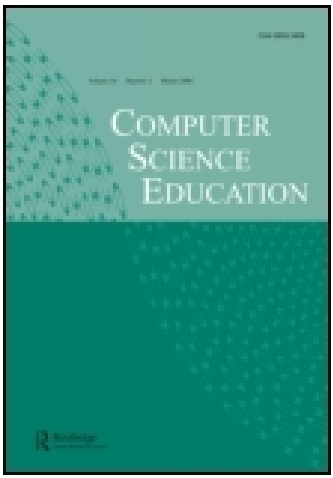

Computer Science Education
Publication details, including instructions for authors and subscription information:
http://www.tandfonline.com/loi/ncse20

Learning to Use Parentheses and Quotes in LISP

Elizabeth A. Davis [a], Marcia C. Linn [a] & Michael Clancy [a]
[a] University of California, Berkeley
Published online: 09 Jul 2006.

COMPUTER SCIENCE EDUCATION 6, 15–31, (1995)

Learning to Use Parentheses and Quotes in LISP

Elizabeth A. Davis
Marcia C. Linn
Michael Clancy
University of California, Berkeley

This article illustrates successful strategies for helping novice programming students understand the use of parentheses (Ps) and quotes (Qs) in LISP. Based on detailed analysis of students working programming problems, we designed lab exercises to address typical difficulties. Rather than emphasizing correct answers, our approach, inspired by the scaffolded knowledge integration (SKI) framework, required students to take the role of investigator and critic. In particular, the intervention required students to critique incorrect calls, identify the incorrect rule being applied in the incorrect call, and correct the call. It also required students to identify causes of error messages we identified as difficult. These exercises had a positive effect not just on the calls to functions, which the intervention specifically addressed, but also on writing function definitions and predicting results, which were not specifically addressed by the intervention. The exercises improved students' understanding of LISP and increased the efficiency of future instruction.

1. INTRODUCTION

Improving programming instruction has proven difficult. Telling students the "right" answer often fails. For many students, a more active

This material is based on research supported by the National Science Foundation under Grant Number MDR–89–54753. Any opinions, findings, and conclusions or recommendations expressed in this publication are those of the authors and do not necessarily reflect the views of the National Science Foundation.

We thank Lydia Mann, Christopher Hoadley, Sherry Hsi, Christina Schwarz, and Joshua Paley for helpful comments and discussion. We appreciate the interest and cooperation of the LISP students. Thanks also go to Dawn Davidson and Madeleine Bocaya for manuscript production assistance.

Correspondence and requests for reprints should be sent to Elizabeth Davis, School of Education, Education in Math, Science, and Technology, 4533 Tolman Hall, University of California at Berkeley, Berkeley, CA 94720.

and guided approach works far better. We addressed difficulties faced by the novice in learning to use parentheses (Ps) and quotes (Qs) in LISP by guiding them as they made discoveries about Ps and Qs. We devised an intervention guided by the scaffolded knowledge integration (SKI) framework [1] that incorporated extensive study of students learning LISP [2–4]. To devise the intervention, we conducted interviews with students in an introductory computer science course for nonmajors. In the interviews, students either (a) worked a long set of problems offline or (b) worked a shorter set of the same problems both offline and then online. All of the problems focused on calling and defining functions and predicting the results. They were designed to draw out incorrect rules we predicted students would have. The intervention also incorporated our experience teaching the LISP and insights from two graduate students learning LISP who discussed their conceptions at weekly research meetings.

We have observed that the difficulties encountered by many novice LISP programmers stem from their basic beliefs regarding placement of Ps and Qs [2–4]. Ps and Qs form the foundation of LISP. Simply put, Ps surround function calls and are used to denote list data structures; Qs indicate that an item following an open P is not meant to be evaluated.

2. BACKGROUND: CONSTRUCTION OF KNOWLEDGE ABOUT Ps AND Qs

Integrated knowledge is knowledge that is linked and connected. Our intervention is based on the SKI framework combined with studies of how students make sense of LISP. Instruction following the SKI framework provides support for students as they make sense of complex material while at the same time encouraging learners to take responsibility for their own learning. The SKI framework has four main tenets. The primary aspect of SKI addressed by the intervention is that of the investigator and critic role played by students. Other studies have addressed in greater detail the remaining three tenets of SKI, which will be reviewed in this article [1].

First, the framework emphasizes instructional goals that build on students' intuitive models and help students expand their repertoire of ideas. With regard to LISP, we studied students' rules for Ps and Qs and designed instruction that expanded their repertoire by offering appealing alternative formulations of those rules [3].

Second, the framework makes thinking visible by engaging students in activities that provide as much support as students need while still encouraging autonomy. For example, students used case studies to make

expert thinking apparent as well as software tools to help explicate the evaluation process used in LISP [5–7].

Third, the SKI framework supports students as they link and connect information, often taking advantage of the social nature of learning [1]. In LISP, students need to link such information as (a) abstract statements in texts, (b) concrete examples in texts and other materials, (c) computer feedback, (d) descriptions in lectures, and (e) discussions in case studies. Observations and investigations indicate that students often find programming explanations vague or incomplete and are prone to rely on examples far more than they rely on descriptions or instructions [8,9]. In our course, students worked in teams during supervised laboratory sessions and helped each other link this information.

Fourth, and most important in this study, the SKI framework engages students as investigators and critics so they can develop skill in self-monitoring and reflection. In working with LISP, students not only solved problems, but also critiqued problem solutions generated by others. The SKI framework supports students as they draw on their previous experience to critique and understand LISP. Although making analogies to other domains is somewhat controversial in programming, it is common in most domains [10]. Anderson, Farrell, and Sauers [11] asserted that there are few analogies from other domains to LISP so students must rely primarily on LISP examples for guidance. Others have suggested that spoken and written languages such as English provide analogies for programming [8,12]. As students critique code, they necessarily build on their previous experiences. Critiquing activities help students to integrate their understanding rather than merely memorizing rules.

In LISP, as in many other domains, integrating ideas is difficult. Courses provide fragmented and possibly contradictory information about programming. Students usually find examples too specific and rules too abstract. To help students acquire more cohesive understanding of programming, Clancy and Linn [13] devised case studies and demonstrated their effectiveness [6]. Even with case studies to guide and support them, students find it difficult to combine the information they encounter and are uncertain about how it should be applied. This challenge applies even to areas experts find simple, such as Ps and Qs. Segal, Ahmad, and Rogers [8] found that novices make errors regarding the syntax and the underlying semantics in too systematic a manner to be merely carelessness (although happily they tend to make such errors less often as they gain more experience with the language). To make matters worse, in addition to needing to make sense of LISP Ps and Qs, students are also faced with confusing and cryptic error messages and computer feedback. Moreover, textbooks rarely address the use of error

messages and computer feedback. Because of this shortcoming, students are left without any guidance as to how to construct integrated knowledge about using this feedback. They often instead disregard the feedback as incomprehensible or irrelevant.

In sum, students are typically confronted with an abstract summary of LISP Ps and Qs, concrete examples of their use, and incomprehensible experiences with computer feedback. To make sense of this information they must somehow combine it and determine how it is applied.

We found that a student's set of beliefs about computers and programming influences how that student makes sense of programming. As novices attempt to make sense of LISP, they inevitably take a stance toward the nature of programming knowledge itself. Studies suggest that the beliefs about knowledge in a given domain influence learning [*14,15*].

Essentially, students viewed the syntax of LISP as governed by rules. Since many of the students studied were taking their first programming course, they drew on their analogous experiences in other disciplines and on their beliefs about knowledge in other disciplines to make sense of programming knowledge. For example, they likely had experience with the syntax of algebra and with English and other spoken languages. As a result they may have believed that syntax is governed by rules that always apply (as in algebra symbol manipulation) or that are made to be broken (as in English grammar).

3. METHOD AND RATIONALE

Previous studies [*2–4*] confirmed our suspicions about the difficulty of Ps and Qs and generated a list of predicted rules that we expected students to use. We identified ways in which students perform in offline and online situations, and we characterized successful and unsuccessful students in those situations. We then asked, "How could we help less successful students become more successful?" This question led us to create a Ps and Qs intervention to assist students as they struggled to understand Ps and Qs. The intervention focused only on the calls of functions, as this was the most problematic area for students. A similar intervention could, and perhaps should, be designed and implemented for function definitions.

3.1 Interviews

We designed two types of interviews to assess students' approaches to using Ps and Qs. *Offline interviews* investigated students' answers to 23 problems focusing on Ps and Qs that students worked exclusively

offline. In *online interviews*, students worked eight problems (a subset of the 23 used in the offline interviews) on paper and then worked some or all of those eight problems on the computer as well. Students were instructed to think aloud during both types of interviews and results were compiled based on the written, oral, and (where appropriate) electronic responses.

Both offline and online assessment environments are considered important. The offline environment provides a larger base from which conclusions may be drawn, but the online environment is one closer to that in which many students claim to prefer to work. Because of the cryptic nature of much of computer feedback, students may abandon the strategy of reflection and instead view error messages as generic [4]. By focusing our attention on assessing students' use of computer feedback and improving their understanding of it, we assisted them in developing useful strategies for various types of computer usage in their future work.

The incorrect rules we identified in the interviews are presented in Figures 1 through 3. The intervention, designed to help students refine their rules or identify new rules, focused on the most prevalent of the call rules in Figure 1, although students were seen to use all of these rules. For example, if a student mistakenly applied the *quote distributed* rule, he or she might type "func-name ('a 'b 'c)" rather than the correct call of "func-name '(a b c)." The letters given below the rule name in this figure correspond to those in the Lab 2 intervention (see the Appendix), which will be fully discussed later. Figures 2 and 3 present the most common incorrect rules used in writing definitions and predicting results of functions, respectively.

3.2 Intervention

We primarily implemented the investigator and critic tenet of SKI to help students make sense of Ps and Qs. Students do create their own LISP rules and find it difficult to construct reliable rules for Ps and Qs. To overcome this problem, we created an intervention with two primary components. In the first part of the intervention, students discover that some rules consistently work and some do not. Students critique each call by identifying the incorrect rules being used and then they correct the calls. When students critique incorrect calls, they discover on their own which rules work. The intervention helps students expand their repertoire of rules and distinguish the most reliable rules. In the second part of the intervention, to help students gain experience with using computer feedback, we have them identify the causes of certain difficult and prevalent error messages. This helps them see the importance of paying attention to and interpreting error messages.

Rule Name	Description of Inappropriate Rule	Examples (other examples are possible)
Arguments Grouped A and B in Lab 2	Encloses all arguments together in ( ) or '( ).	**Correct call:** (add-lists '(1 2 3) '(9 8 7)) **Call with rule applied:** (add-lists '((1 2 3) (9 8 7)))
Lists Unquoted C and D in Lab 2	Does not quote lists.	**Correct call:** (add-lists '(1 2 3) '(9 8 7)) **Call with rule applied:** (add-lists (1 2 3) (9 8 7))
Non-numeric Atoms Unquoted E in Lab 2	Does not quote non-numeric atoms when appropriate to do so.	**Correct call:** (make-a-sentence 'cow 'jumps 'brown) **Call with rule applied:** (make-a-sentence cow jumps brown)
Quote Distributed F in Lab 2	Uses quotes inside of list, before each element, instead of before opening parenthesis.	**Correct call:** (add-lists '(1 2 3) '(9 8 7)) **Call with rule applied:** (add-lists ('1 '2 '3) ('9 '8 '7))
Non-numeric Atoms Misrepresented	Encloses individual non-numeric atoms in ( ) or '( ).	**Correct call:** (make-a-sentence 'cow 'jumps 'brown) **Call with rule applied:** (make-a-sentence '(cow) '(jumps) '(brown))
Numbers Misrepresented	Encloses individual numeric atoms in ( ) or '( ).	**Correct call:** (add-and-mult 2 2 4) **Call with rule applied:** (add-and-mult (2) (2) (4))
Nested Functions Misused	Exhibits general difficulty with nested functions. This might include quoting the call to the nested function but still evaluating it, ignoring appropriate syntax for the arguments of the nested function, or any of several other possibilities.	**Correct call:** (equal-second-elem? '(1 2 3) (new-first 2 '(a b c))) **Call with rule applied:** (equal-second-elem? '(1 2 3) new-first 2 a b c)

Figure 1. Inappropriate call rules identified.

Rule Name	Description of Inappropriate Rule	Examples (other examples are possible)
Function Misused in Listing	Treats function that is to return a list strangely, for example by quoting it or quoting a value to be returned.	**Correct expression:** (rest L) **Expression with rule applied:** '(rest L)
Variable Misused in Listing	Treats variable representing a list strangely, for example by quoting it, parenthesizing it, or using some other incorrect syntax.	**Correct expression:** (if (equal (first L) (second L)) (rest L) L) **Expression with rule applied:** (if (equal (first L) (second L)) (rest L) (list L))
Parentheses Misused in Listing	Creates list by using parentheses.	**Correct expression:** (list (first L) (second L)) **Expression with rule applied:** ((first L) (second L))
Arguments Grouped in Definitions	Groups arguments to internal function calls.	**Correct expression:** (list x y z) **Expression with rule applied:** (list (x y z))
Variables Problems	Encloses individual variables in parentheses or quotes variable names.	**Correct expression:** (list x y z) **Expression with rule applied:** (list (x) (y) (z))
Header Problems	Displays problems in writing header, for example uses quotes in header to denote a list, uses incorrect parentheses, or uses placeholder for each element of list in header.	**Correct expression:** (defun mult-all-by-n (n L) **Expression with rule applied:** (defun mult-all-by-n (n '(x y z))
Parentheses Missing	Does not use parentheses around internal functions.	**Correct expression:** (list (first L) (second L)) **Expression with rule applied:** (list first L second L)
Special Forms Misused	Uses incorrect constructs for special forms, particularly "if" and "cond."	**Correct expression:** (if (equal (first L) (second L)) (rest L) L) **Expression with rule applied:** (if (equal (first L) (second L) ((rest L) (list L))))
Internal Functions Quoted	Quotes internal functions but still evaluates them.	**Correct expression:** (list (first L) (second L)) **Expression with rule applied:** (list '(first L) '(second L))
Definition Wrong	Writes incomplete or incorrect definition.	

Figure 2. Inappropriate definition rules identified.

Specifically, all of these exercises helped students explore how LISP evaluation works, rather than relying on trial-and-refinement strategies as had students in previous semesters. Students participated in the intervention during lab assignments given in the 2nd and 3rd weeks of class. In Lab 2, students critiqued incorrect function calls and several rules known to be most problematic for students. They identified the rule or rules being applied and corrected the call. They also created several function calls, given a function definition and a desired result for each. In Lab 3, students predicted the result or error message a call would return and explained the reasons for the message. The full intervention appears in the Appendix.

Rule Name	Description of Inappropriate Rule	Examples (other examples are possible)
Missing or Extra Parentheses	Returns output with missing, extra, or unbalanced parentheses.	**Correct result:** (2 B C) Result with rule applied: 2 B C
Output Quoted	Includes quotes in front of values that are returned via the quote mechanism.	**Correct result:** NO-MATCH Result with rule applied: 'NO-MATCH
Output Incorrect	Returns wrong output for given function.	**Correct result:** NIL **Result with rule applied:** 24

Figure 3. Inappropriate result rules identified.

3.3 Design

We evaluated the intervention by comparing performance of fall semester students who did not use the intervention to spring students who did. The fall semester served as a baseline for comparison with the spring semester.

The two groups studying LISP were similar. There was neither a significant difference between the overall student populations' GPAs or year in school in the two compared semesters, nor were there any major differences in the curriculum or exercises other than the inclusion of the intervention. The same professor (Clancy) has taught the course for four consecutive semesters.

4. RESULTS AND DISCUSSION

Several forms of evidence confirmed the effectiveness of the intervention. First, anecdotal evidence from interviews demonstrated that students' online performance in the intervention semester exceeded that in the baseline semester. Second, analysis of the number of completely correct calls and number of errors (inappropriate rules used) on the Ps and Qs assessment indicates that there was a large improvement on Ps and Qs in the intervention semester compared with the baseline semester. A gain from 31 to 40 completely correct calls, $F\ (1, 68) = 9.38$, $p < .01$, and a decline from 25 to 14 total errors, $F\ (1, 68) = 6.14$, $p < .01$, occurred.

Students made a significantly higher number of completely correct calls and a significantly lower number of total errors. In an earlier study, we defined "successful" students who used well-refined rules and "other" students who either did not use rules or were seeking rules without refining them. During the baseline semester, 75% of all students were in the successful group identified in our previous work [3], while in the intervention semester this number rose to 88%. Thus, the intervention resulted in overall improved performance.

Figure 4 summarizes the primary rules displayed by the students in the baseline and intervention semesters, with two scaled numbers for each rule. The first number represents the number of occurrences of the rule divided by the number of subjects in that semester's interviews. The second number represents the number of subjects who used the rule, again divided by the total number of subjects.

Almost all of the incorrect rules were used less in the intervention semester than they were in the baseline semester. *Lists unquoted* is dramatically reduced for the intervention students, while *output incorrect* and *special forms misused* become primary result and definition rules for

Primary Rule	Number of Occurrences (Scaled by number of subjects)		Number of People (Scaled by number of subjects)	
	Baseline	Intervention	Baseline	Intervention
Call Rules:				
Arguments Grouped	3.89	1.79	1.31	1.09
Lists Unquoted	4.33	0.65	1.86	0.41
Nested Functions	1.36	0.74	1.14	0.71
Definition Rules:				
Variable Misused in Listing	0.50	0.53	0.50	0.53
Special Forms Misused	0.25	0.59	0.25	0.44
Result Rules				
Missing or Extra Parentheses	2.61	1.59	1.28	1.00
Output Incorrect	1.75	2.21	0.97	1.06

Figure 4. Comparison of primary rules used, baseline semester versus intervention semester.

the intervention students. The number of wrong definitions went down in the intervention semester, indicating greater overall understanding of LISP and its evaluation with the intervention.

Understandably, the primary difference between the semesters is found in the call rules because the intervention focused directly on calls. The intervention specifically addressed the *arguments grouped, lists unquoted, numeric lists unquoted, atoms unquoted,* and *quote distributed* rules. Thus, these rules in general can be viewed as improving because of the intervention. Lists unquoted and numeric lists unquoted, which had been primary rules in the baseline semester, decreased dramatically. However, arguments grouped remains a problem even though it was addressed by the intervention, and did, in fact, decrease as well.

Reasonably, it appears that as people gain experience and solve problems making them think about how LISP's Ps and Qs are applied, they get a better understanding of what LISP returns because they better understand the evaluation process. For example, they have more of an idea of what will be returned in Ps and what will not be. However, the intervention results indicate that slightly more people returned incorrect output than had in the baseline semester. People seemed to be better at understanding LISP evaluation, but this improvement came at the expense of performance in, say, the arithmetic the function applied, perhaps as the result of an increased cognitive load.

Even with the intervention, not all students developed an integrated understanding of Ps and Qs. Some students remained in the less successful categories we have identified. The students who did not appear to benefit at all from the intervention are those who did not reflect on their answers—for example, simply rewording an error message when asked to explain it. These students did not appear to put effort into the particular lab exercises comprising the intervention.

Nonetheless, the intervention continues to be used in versions of the introductory course currently being taught, and it is by and large successful. Although we have not continued our deep analysis of its effect, anecdotal evidence suggests that more students continue to understand the use and meaning of Ps and Qs than they had before the intervention was included in the course.

5. IMPLICATIONS FOR INSTRUCTION

We recommend that teachers of other introductory LISP courses incorporate similar exercises to improve their students' understanding of evaluation. Furthermore, we recommend that teachers of introductory

courses of other languages make their basics more explicit, as well. (For example, Linn and Clancy [6] showed that case studies based on templates helped novices understand Pascal.)

Our Ps and Qs intervention improved most students' understanding of LISP function calls and definitions and LISP evaluation in general. Overall student performance in the interview was significantly better in the intervention semester than it was in the baseline semester. The intervention hastened and improved the refinement process. In other words, some of the students who would have refined their rules anyway were scaffolded in doing it sooner, and some of those who were seeking rules seemingly at random were encouraged to see consistency in their work. Likewise, students who had already started to make sense of computer-generated error messages were encouraged to continue to do so, while those who were previously seeing error messages as generic were exposed explicitly to the differences.

The intervention primarily implemented the critiquing part of the SKI framework. Facilitating students' reflection on what they are doing and why they are doing it was successful. While explicitly pointing out "wrong" rules is risky, because students could remember the visual appearance of the incorrect rule without remembering that it was wrong, we believe that the intervention was as successful as it was because it encouraged reflection.

Students seek rules to interpret the abstract explanations they see and hear. By encouraging students to evaluate and critique inadequate rules we help them to move along in their refinement process. And, by encouraging them to predict error messages and results, we stress the importance of feedback. Even though the intervention focused only on calls, it seemed to affect students' overall understanding of LISP and its evaluation.

The intervention could be further improved. Some students still exhibited a distressing lack of integrated understanding even though most were significantly more competent at a superficial level. Students do not always understand terms used in instruction such as *evaluation* and even *atoms* and *lists*. Without understanding these basic building blocks of LISP, they are unlikely to develop a full understanding of why Ps and Qs are used the way they are. Increased experience in making sense of LISP building blocks is needed to ensure student understanding of these concepts. Such a supplement should occur even earlier than the Ps and Qs intervention.

Computer feedback is one of the only sources of feedback to which students have regular access. If that feedback is cryptic, they will either struggle (successfully or unsuccessfully) to understand it or in fact may give up and resort to the kinds of strategies they perceive as more use-

ful, such as perceptual matching and asking their not necessarily knowledgeable neighbor. Our research shows that students have significantly more trouble with cryptic error messages and often begin to view error messages as irrelevant and generic when they do not understand them [4]. By making these messages accessible, we provided students with techniques for guiding their own learning.

Instruction in introductory computer science courses, then, should not discount the trouble that students have with aspects of the domain that experts view as simple. These results indicate that acknowledging that students do have difficulty with the basics—here, Ps and Qs and interpreting computer feedback—and addressing them explicitly and early in the course will help to alleviate students' difficulty in these areas and will help to scaffold their knowledge integration about the entire domain.

Asking students to critique rules we know to be problematic, discussing the importance of reading error messages, and explaining how to do so should all help to mitigate some of the difficulties that students have. Encouraging students to identify specific problems with calls and definitions and asking them to predict error messages spawned by incorrect expressions will scaffold them as they make sense of the most basic concepts of a language's content. These tasks were accomplished with our intervention and should be addressed by any introductory course. Our intervention does not fully address, however, students' beliefs about the consistency of computers. After the intervention, some continued to believe that computers are capricious. Introductory programming instruction must develop strategies for addressing this issue as well as the content currently emphasized.

REFERENCES

[1] M. C. Linn, "Designing Computer Learning Environments for Engineering and Computer Science: The Scaffolded Knowledge Integration Framework," *Journal of Science Education and Technology*, Vol. 4, No. 2, pp. 103–125, 1995.

[2] E. A. Davis, "A Computer-Based Diagnostic for Basic LISP Concepts," University of California, Berkeley, CA, Tech. Rep., 1992.

[3] E. A. Davis, M. C. Linn, L. M. Mann, and M. J. Clancy, "Mind Your Ps and Qs: Using Parentheses and Quotes in LISP," in *Empirical Studies of Programmers: Fifth Workshop*, C. R. Cook, J. C. Scholtz, and J. C. Spohrer, Eds. Norwood, NJ: Ablex, 1993, pp. 62–85.

[4] E. A. Davis, M. C. Linn, and M. J. Clancy, "Students' Off-line and On-line Experiences," *Journal of Educational Computing Research*, Vol. 12, No. 2, pp. 109–134, 1995.

[5] J. E. Bell, M. C. Linn, and M. J. Clancy, "Knowledge Integration in Introductory Programming: CodeProbe and Interactive Case Studies," *Interactive Learning Environments*, Vol. 4, No. 1, pp. 75–95, 1994.

[6] M. C. Linn and M. J. Clancy, "The Case for Case Studies of Programming Problems," *Communications of the ACM*, Vol. 35, No. 3, pp. 121–132, 1992.
[7] L. M. Mann, M. C. Linn, and M. J. Clancy, "Can Tracing Tools Contribute to Programming Proficiency? The LISP Evaluation Modeler," *Interactive Learning Environments*, Vol. 4, No. 1, pp. 96–113, 1994.
[8] J. Segal, K. Ahmad, and M. Rogers, "The Role of Systematic Errors in Developmental Studies of Programming Language Learners," *Journal of Educational Computing Research*, Vol. 8, No. 2, pp. 129–153, 1992.
[9] J. R. Anderson, F. G. Conrad, and A. T. Corbett, "Skill Acquisition and the LISP Tutor," *Cognitive Science*, Vol. 13, No. 4, pp. 467–505, 1989.
[10] A. Newell, *Unified Theories of Cognition*. Cambridge, MA: Harvard University Press, 1990.
[11] J. R. Anderson, R. Farrell, and R. Sauers, "Learning to Program in LISP," *Cognitive Science*, Vol. 8, No. 2, pp. 87–129, 1984.
[12] E. Soloway and K. Ehrlich, "Empirical Studies of Programming Knowledge," *IEEE Transactions on Software Engineering*, Vol. 10, No. 5, pp. 595–609, 1984.
[13] M. J. Clancy and M. C. Linn, "Functional Fun," *Special Interest Group on Computer Science Education (SIGCSE) Bulletin*, Vol. 22, No. 1, pp. 63–67, 1990.
[14] N. B. Songer and M. C. Linn, "How Do Students' Views of Science Influence Knowledge Integration?," *Journal of Research in Science Teaching*, Vol. 28, No. 9, pp. 761–784, 1991.
[15] M. C. Linn and N. B. Songer, "How Do Students Make Sense of Science?," *Merrill-Palmer Quarterly*, Vol. 39, No. 1, pp. 47–73, 1993.

APPENDIX: THE Ps AND Os INTERVENTION

Incorrect Rules for Ps and Qs for Use in Part 1

Students sometimes invent incorrect rules for using Ps and Qs. Here are some examples. Each example includes the description of the rule, an example—an incorrect call to a function—of the rule, the correct call, and a framework for the definition of the function to be called.

rule	*explanation*	*incorrect call resulting from following the rule*	*correct call*	*function definition*
A	All inputs provided for a function are enclosed in parentheses and quotes.	(example1 '((a b c) thing)))	(example1 '(a b c) 'thing)	(defun example1 (L1 element) ...)
B	All inputs provided for a function are enclosed in parentheses.	(example2 ((a b c) thing)))	(example2 '(a b c) 'thing)	(defun example2 (L1 element) ...)
C	Lists don't need to be quoted.	(example3 (a b c) (d 2 f))	(example3 '(a b c) '(d 2 f))	(defun example3 (oneList twoList) ...)
D	Lists of numbers don't need to be quoted.	(example4 (5 18 299))	(example4 '(5 18 299))	(defun example4 (numList) ...)
E	Atoms don't need to be quoted.	(example5 try this out)	(example5 'try 'this 'out)	(defun example5 (atom1 atom2 atom3) ...)
F	Quotes may go either inside or outside a list.	(example6 ('this 'is 'a 'list))	(example6 '(this is a list))	(defun example6 (L) ...)

Worksheet for Lab Assignment 3

Consider the following function definitions.

```
(defun first-increased (L num)
  (cons (+ (first L) num) L))

(defun arith-table (num 1 num 2)
  (list (+ num2 num1) (-num2 num1) (* num2 num1)))
```

Part A

What is the result of evaluating the following expression?

```
(first-increased (arith-table 3 4) 22)
```

If evaluation results in an error, give the function whose application produced the error and explain why the error happened.

Part B

What is the result of evaluating the following expression?

```
(first-increased '(arith-table 3 4) 22)
```

If evaluation results in an error, give the function whose application produced the error and explain why the error happened.

Worksheet for Problems in Part 1

Problem 1

function definition	*incorrect call*	*desired result*	*incorrect rules used*	*call that correctly produces the desired result*
(defun first-replaced (item L) (cons item (rest L)))	(first-replaced '(A (THE DOG)))	(A DOG)		
(defun paired-list (L1 L2) (list (list (first L1) (first L2)) (list (second L1) (second L2)) (list (third L1) (third L2))))	(paired-list (3 7 9) ('A 'F 'G))	((3 A) (7 F) (9 G))		
(defun square-then-cons (num L) (cons (* num num) L))	(square-then-cons (4 (2 C F 3)))	(16 (2 C F 3))		

Problem 2

function definition	*desired result*	*call that correctly produces the desired result*
(defun sub-from-all (L num) (list (– (first L) num) (– (second L) num) (– (third L) num)))	(7 4 1)	
(defun pyramid-lists (e1 e2 e3) (list e1 (list e2) e3))	(A (B) C)	
(defun all-firsts (L1 L2 L3) (list (first L1) (first L2) (first L3)))	(A (B) C)	